# CIS 345 Project 2

| | |
|---|---|
| Charles Thompson<br>2821909<br><br>ccthomps | |
| Julie White<br>2832202 | |

## Design and implementation

### Task #1: Efficient Alarm Clock

In pintos, a thread can call timer_sleep() to put itself to sleep for the specified amount of time in ticks. The previous implementation of timer_sleep() was inefficient because it simply called thread_yield() in a loop which puts the thread back on the ready queue. Since it would be scheduled again before it is time to wake up and then yield again, it creates unnecessary overhead.

Our task was to implement a new list, sleeping_list, where the thread would sit until it is time to wake up. When a thread calls timer_sleep(), it calculates the absolute wakeup time and passes that to a new function thread_sleep() which:

1. Changes the thread's state to THD_SLEEP
2. Sets a new member of the thread struct called wake_up_time to the absolute wake up time in ticks
3. Inserts the thread into a sleeping list ordered by the absolute wake up time
4. Yields the CPU to another thread

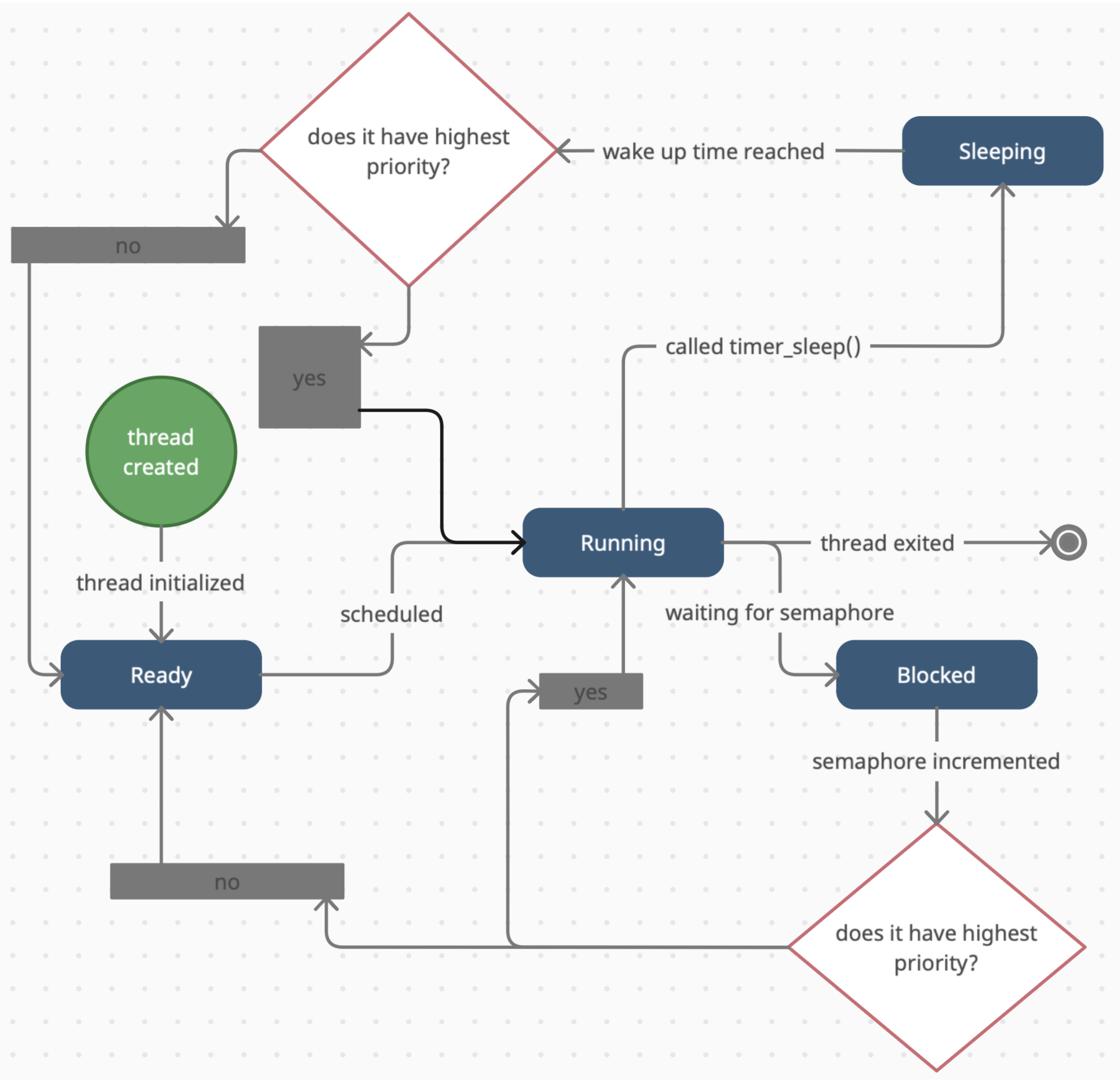
To wake these threads up when it's time, we take advantage of the timer interrupt which happens every tick. The timer interrupt handler now calls a new function, thread_wake_up(), which:

1. Removes any threads from the sleeping list which need to wake up
2. Places those threads back on the ready queue and sets their state to THD_READY
3. Checks the priority of the threads woken up and preempts the running thread if needed

### Task #2: Basic Priority Scheduling

Pintos already provided an interface for threads to set their priority, however the scheduler did not respect these values. The old scheduler used a FIFO ready list. Our new implementation uses an ordered list such that the highest priority thread is always up next. This alone is not sufficient, as threads that have been unblocked may need to preempt the running thread. To solve this, we changed the waiting list of semaphores to be an ordered list as well. Then when a thread increments a semaphore, the waiter with the highest priority will be unblocked. We then check this thread's priority and preempt the running thread if needed. No additional work is needed to ensure threads that were waiting for a lock can preempt the running thread as lock_release() already calls sema_up(). Lastly, newly created threads may need to preempt the running thread if the new thread has a higher priority. Therefore in thread_create(), we check for this condition and yield if so.

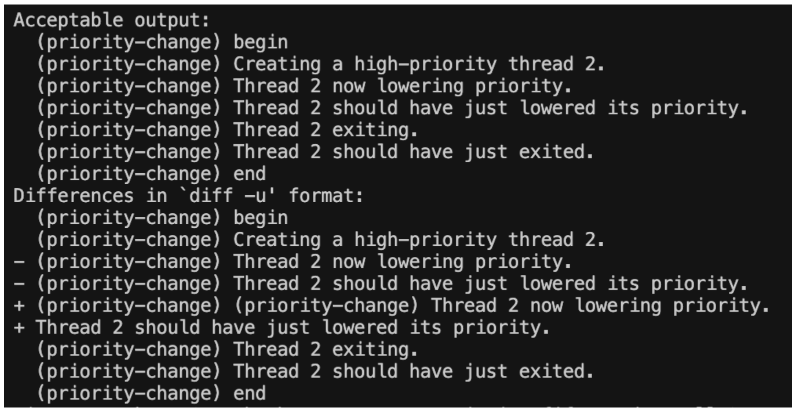
# State Transition Diagram

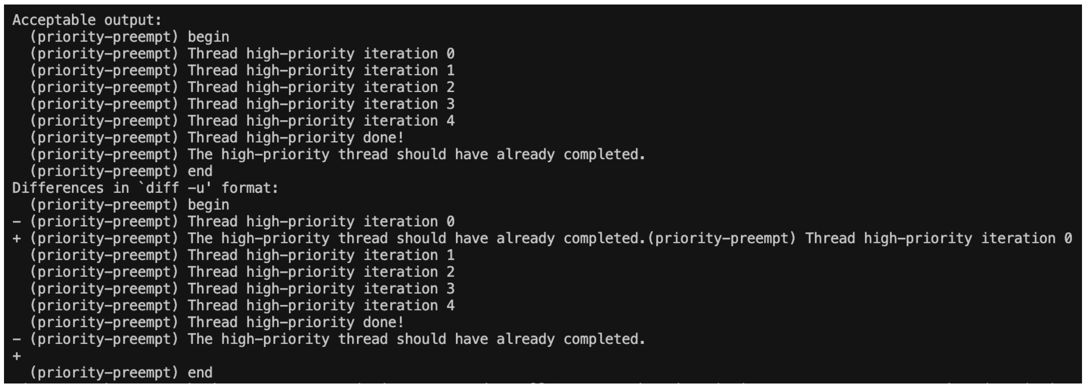


## Testing/Project status

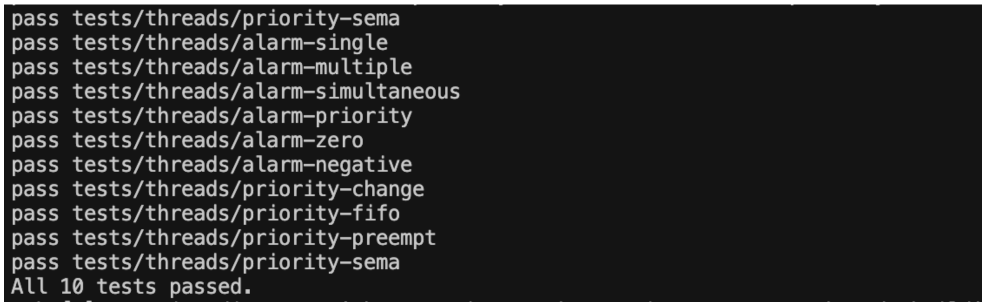
When beginning this project, the first hurdle we encountered was fully understanding the Pintos environment.  We needed to understand the functions being called and where they were being called from for each test to be fully able to complete this project.  We spent a lot of time just familiarizing ourselves with the Pintos operating system.

The biggest issue that we ran into was that we struggled to get two tests to pass.  These tests (priority-change and priority-preempt) continually failed based on incorrect printing.  This was difficult to figure out. Based on the first image (priority-change), we assumed it had something to do with the way Pintos handles the output buffer and that sorting the semaphore messed it up. But the second image (priority-preempt) is what gave it away.


```
Acceptable output:
  (priority-change) begin
  (priority-change) Creating a high-priority thread 2.
  (priority-change) Thread 2 now lowering priority.
  (priority-change) Thread 2 should have just lowered its priority.
  (priority-change) Thread 2 exiting.
  (priority-change) Thread 2 should have just exited.
  (priority-change) end
Differences in `diff -u' format:
  (priority-change) begin
  (priority-change) Creating a high-priority thread 2.
- (priority-change) Thread 2 now lowering priority.
- (priority-change) Thread 2 should have just lowered its priority.
+ (priority-change) (priority-change) Thread 2 now lowering priority.
+ Thread 2 should have just lowered its priority.
  (priority-change) Thread 2 exiting.
  (priority-change) Thread 2 should have just exited.
  (priority-change) end
```


```
Acceptable output:
  (priority-preempt) begin
  (priority-preempt) Thread high-priority iteration 0
  (priority-preempt) Thread high-priority iteration 1
  (priority-preempt) Thread high-priority iteration 2
  (priority-preempt) Thread high-priority iteration 3
  (priority-preempt) Thread high-priority iteration 4
  (priority-preempt) Thread high-priority done!
  (priority-preempt) The high-priority thread should have already completed.
  (priority-preempt) end
Differences in `diff -u' format:
  (priority-preempt) begin
- (priority-preempt) Thread high-priority iteration 0
+ (priority-preempt) The high-priority thread should have already completed.(priority-preempt) Thread high-priority iteration 0
  (priority-preempt) Thread high-priority iteration 1
  (priority-preempt) Thread high-priority iteration 2
  (priority-preempt) Thread high-priority iteration 3
  (priority-preempt) Thread high-priority iteration 4
  (priority-preempt) Thread high-priority done!
- (priority-preempt) The high-priority thread should have already completed.
+
  (priority-preempt) end
```

The test creates a higher priority thread and in the next line, prints that it should have finished. That print statement actually executed before the thread finished. We then realized that if a thread creates a thread with a higher priority, the newly created thread should preempt the first. Therefore in thread_create(), we check for this condition and yield if so. This solved the issue.

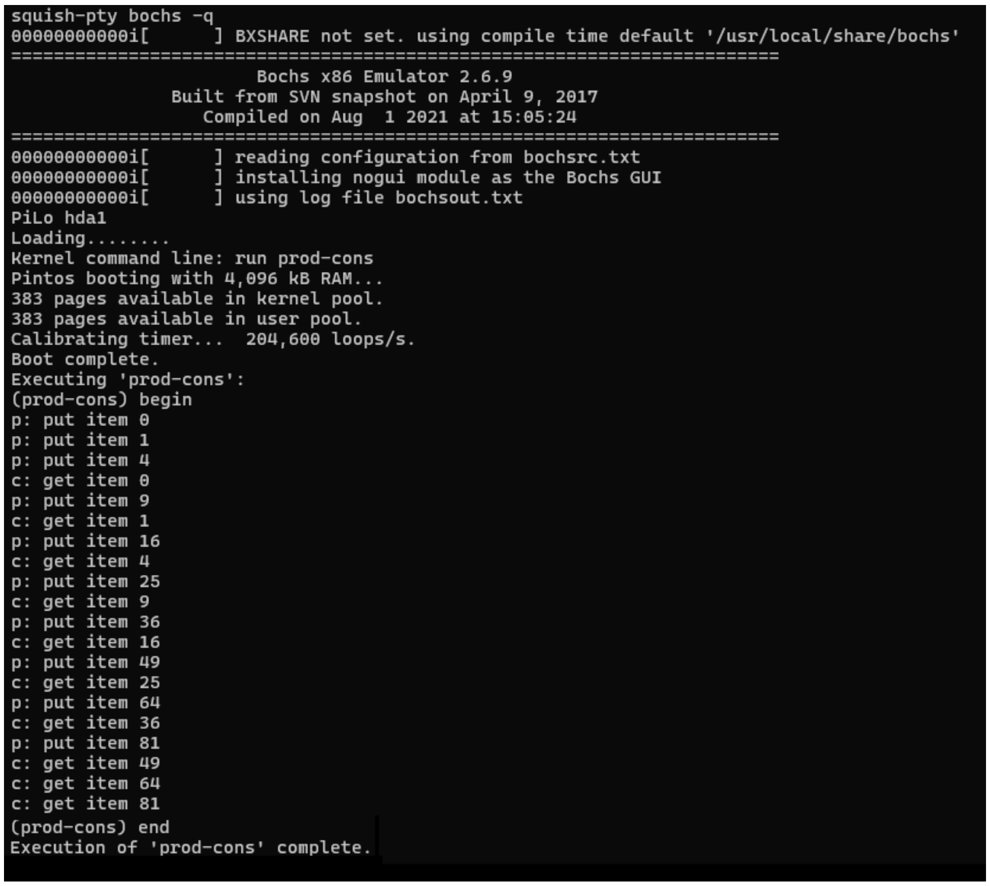

```
pass tests/threads/priority-sema
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
All 10 tests passed.
make[1]: Leaving directory '/home/student/ccthomps/pintos_csu/src/threads/build'
```

We believe that the incorrect printing was because the test thread was preempted by the new thread during the timer interrupt rather than during the thread creation.

Solving this issue was made easier by the output printing in “diff -u” format. This allowed us to easily spot the differences between the expected and actual outputs. Giving us the ability to pinpoint the exact moments where the problem occurred and helping us to figure out what was going wrong.

## Producer and Consumer Tests

The first of the producer/consumer tests that we ran was making the producer have higher priority than the consumer (shown below).

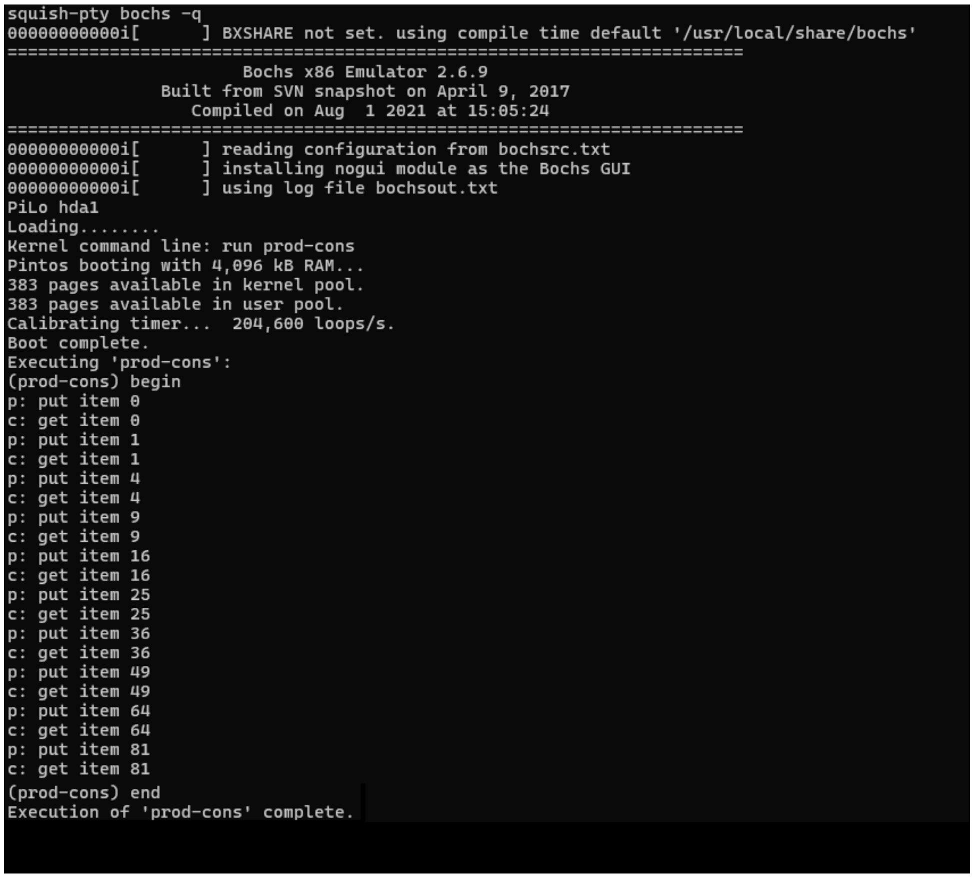


```
squish-pty bochs -q
00000000000i[      ] BXSHARE not set. using compile time default '/usr/local/share/bochs'
========================================================================
                       Bochs x86 Emulator 2.6.9
               Built from SVN snapshot on April 9, 2017
                  Compiled on Aug  1 2021 at 15:05:24
========================================================================
00000000000i[      ] reading configuration from bochsrc.txt
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading........
Kernel command line: run prod-cons
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
Boot complete.
Executing 'prod-cons':
(prod-cons) begin
p: put item 0
p: put item 1
p: put item 4
c: get item 0
p: put item 9
c: get item 1
p: put item 16
c: get item 4
p: put item 25
c: get item 9
p: put item 36
c: get item 16
p: put item 49
c: get item 25
p: put item 64
c: get item 36
p: put item 81
c: get item 49
c: get item 64
c: get item 81
(prod-cons) end
Execution of 'prod-cons' complete.
```

The output we received is consistent with what was expected. The producer with higher priority will continue to put items into the buffer until the buffer is at capacity. At this point, the consumer will take an item from the buffer, meaning it is no longer at capacity. This unblocks the producer. Given the producer has the highest priority, it will put another item to the buffer and max out capacity again, leaving the consumer to take one item. This will continue to alternate until the producer has run out of items to put into the buffer. At this time, the consumer can take the items in the buffer to finish off the execution.

Because the producer has a higher priority, if there is space in the buffer, the producer will always take priority and put another item in the buffer.  It isn't until the buff is at capacity that the consumer gets to act.

The second producer/consumer test that we ran was making the consumer higher priority over the producer (screenshot below).



```
squish-pty bochs -q
00000000000i[      ] BXSHARE not set. using compile time default '/usr/local/share/bochs'
========================================================================
                        Bochs x86 Emulator 2.6.9
                Built from SVN snapshot on April 9, 2017
                   Compiled on Aug  1 2021 at 15:05:24
========================================================================
00000000000i[      ] reading configuration from bochsrc.txt
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading........
Kernel command line: run prod-cons
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
Boot complete.
Executing 'prod-cons':
(prod-cons) begin
p: put item 0
c: get item 0
p: put item 1
c: get item 1
p: put item 4
c: get item 4
p: put item 9
c: get item 9
p: put item 16
c: get item 16
p: put item 25
c: get item 25
p: put item 36
c: get item 36
p: put item 49
c: get item 49
p: put item 64
c: get item 64
p: put item 81
c: get item 81
(prod-cons) end
Execution of 'prod-cons' complete.
```

Again, the output we received made sense.  The consumer cannot take an item from the buffer that is not there, so the producer has to put an item into the buffer first.  However, as soon as there is an item in the buff to be taken, the consumer takes it (as it has higher priority).  So, the threads consistently alternate– the producer puts an item into the buffer and the consumer takes it.  This continues until items have been put into/taken out of the buffer.

Because the consumer has higher priority, if there is an item in the buffer, the consumer will take priority and take that item from the buffer.